Efficient Generated Libraries for Asynchronous Derivative Computation

D. Buntinas* A. J. Malozemoff† J. Utke*‡

Abstract

The computation of derivatives via automatic differentiation is a valuable technique in many science and engineering applications. While the implementation of automatic differentiation via source transformation yields the highest-efficiency results, the implementation via operator overloading remains a viable alternative for some application contexts, such as the computation of higher-order derivatives or in cases where C++ still proves to be too complicated for the currently available source transformation tools. The Rapsodia code generator creates libraries that overload intrinsics for derivative computation. In this paper, we discuss modifications to Rapsodia to improve the efficiency of the generated code, first via limited loop unrolling and second via multithreaded asynchronous derivative computation. We introduce the approaches and present runtime results.

1 Introduction

Computing derivatives of numerical models $\boldsymbol{f}(\boldsymbol{x}) \mapsto \boldsymbol{y} : I\!R^n \mapsto I\!R^m$, given as a computer program P, is an important but also computation-intensive task. Automatic differentiation (AD) [1] provides the means to obtain such derivatives and is used in many science and engineering contexts (refer to the recent conference proceedings [2, 3] and the AD community website [4]).

Two major groups of AD tool implementations are source transformation tools and operator overloading tools. Among the noteworthy examples of the latter group are Adol-C [5] and HSL_AD02 [6]. Both provide the capability to compute higher-order derivatives. This computation is done by overloading the operators (e.g., `+`, `*`, `/`) and intrinsic functions (e.g., `sin`, `sqrt`) in C++ and Fortran, respectively, for an *active* type. In a simplified view, the active type for a scalar program variable `v` is the vector of coefficients $\boldsymbol{v} = [v_0, v_1, \ldots, v_o]$, up to a certain order o, of the Taylor polynomial $v_0 + v_1 h + v_2 h^2 + \ldots + v_o h^o$. For each of the operators and intrinsic functions one can derive a procedure

*Argonne National Laboratory
†McGill University
‡University of Chicago

```
1  r.v = a.v * b.v;
2  r.d1_1 = a.v * b.d1_1 + a.d1_1 * b.v;
3  r.d1_2 = a.v * b.d1_2 + a.d1_1 * b.d1_1 + a.d1_2 * b.v;
4  r.d1_3 = a.v * b.d1_3 + a.d1_1 * b.d1_2 + a.d1_2 * b.d1_1 + a.d1_3 * b.v;
5  r.d2_1 = a.v * b.d2_1 + a.d2_1 * b.v;
6  r.d2_2 = a.v * b.d2_2 + a.d2_1 * b.d2_1 + a.d2_2 * b.v;
7  r.d2_3 = a.v * b.d2_3 + a.d2_1 * b.d2_2 + a.d2_2 * b.d2_1 + a.d2_3 * b.v;
```

Figure 1: Generated code for overloaded `*` operators with o=3, d=2, for `r=a*b` and $\boldsymbol{v}^j = [v.\mathtt{v}, v.\mathtt{d}j\_\mathtt{1}, \ldots, v.\mathtt{d}j\_\mathtt{3}]$.

that computes the Taylor coefficients of the intrinsic's result from the Taylor coefficients of its arguments. For example, for the multiplication `w=u*v` it is

$$w_k = \sum_{j=0}^{k} u_j \cdot v_{k-j}, \text{ for } k = 0, \ldots, o \quad .$$

The code in the overloaded operators implements the logic for the propagation of the Taylor coefficients. Because of the relatively high complexity of implementing and efficiently using the *reverse* mode of AD compared to the *forward* mode (see [1]), one assumes that for cases where there are few independents relative to the number of dependents in $\boldsymbol{f}$ or in applications that need higher-order derivatives ($o \geq 3$), the forward mode is appropriate. Without further explanation we assume from here on the use of forward mode. A principal ingredient for the efficient computation of higher-order tensors is the propagation of Taylor coefficients in multiple, preselected directions d followed by an interpolation to compute the tensor elements as described in [7]. There, the first coefficients x_1^j related to the n inputs of $\boldsymbol{f}$ form a *seed matrix* $\boldsymbol{S} \in I\!R^{n \times d}$ and the higher-order coefficients are set to zero. In the *vector forward* mode, the active type may be viewed as a collection of coefficient vectors $\boldsymbol{v}^j, j = 1, \ldots, d$. The propagation logic, as in the example for `w=u*v` above, is therefore wrapped in an outer loop over the directions. In the implementation of Adol-C and HSL_AD02 these loops can be found in the body of the overloaded operators.

Based on the observation that, for fixed o and d, unrolling these loops in the code often leads to a performance advantage, the Rapsodia library generator was developed [8, 9]. A large number of operators and intrinsics are common to both C++ and Fortran. Therefore, the code generator was designed to be able to create both C++ and Fortran libraries based on common abstract syntax trees for the operators and intrinsics. An example for the body of an unrolled overloaded `*` operator can be found in Fig. 1. The value `v.v`($\equiv v_0$) of the original program variable is shared among all directions. A convenient side effect of the code generation is that the exploding number of overloading variants[1] that need to be defined is covered as well. The Rapsodia manual [10] provides details and examples for the use of the generator. The application of Rapsodia to practical

[1] The operators/intrinsics need to be defined for all possible combinations of active and passive arguments of different type and precision, including the `complex` type in Fortran.

problems follows the approach that is known, for example, from Adol-C, and therefore we will not allude to it in this paper. In various test scenarios with a variety of compilers and optimization flags, one can observe speedups of up to 10 over a reference implementation; see [9]. Some uses of the higher-order derivatives computed with Rapsodia are described in [11, 9].

2 Motivation

Despite the sizable speedup factors observed for the Rapsodia-generated code, the strategy of completely unrolling the loops has limitations. The strategy is successful for small o and d in part because it permits a *flat data structure* for the active type; that is, the program variables for Taylor coefficients have a fixed offset at compile time, rather than an offset computed based on the loop indices, which are computed at run time. Thus, completely unrolling loops aids compiler optimization. While it would be hard to provably quantify the eventual speedup originating from this particular strategy of generating code, we deem it to be essential. Even for moderate o and d, however, the size of the generated propagation code grows to a point that—combined with the aforementioned inflation of overloading variants—causes very long compile times. The compile time increase is most apparent when compiler optimization is set to high levels[2], while we observed diminished speed advantages for large o and d. Even without access to the internals of the compilers, we can be certain that the code explosion negatively impacts the compiler optimization algorithms (e.g., register allocation) and that it is the root cause for the diminished performance. Thus, some limit to the loop unrolling should be beneficial by reducing the code size while retaining some of the advantages for larger o and d. In Sec. 3 we discuss a simple approach and present some results.

Another avenue for improving the efficiency of the derivative computation is to utilize the availability of multicore hardware. Several forays have already been made in that direction; see, for instance, [12, 13, 14], most of which use OpenMP. In Sec. 4 we describe the problems we experienced with the use of OpenMP and the alternative implementation that employs a queue to asynchronously compute the derivatives with pthreads and with the help of the OpenPA library [15].

3 Modifying Rapsodia to Limit the Unrolling of Loops

The principal elements of the computation common to Taylor propagation logic for overloaded operators can be characterized as follows.

- An outer loop over the directions $i = 1, \ldots, d$.

[2] In some cases tests were aborted because the compiler did not finish within 30 minutes.

- One or more inner loops within the outer loop over the order $k = 0$ (or 1), $\ldots, o$. Cases where there is more than one inner loop result from scaling coefficients before or after the propagation logic; this is done, e.g., for the intrinsics `s=sin(u)` and `c=cos(u)`, where the coefficients are computed together as

$$\tilde{s}_k = \sum_{j=1}^{k} \tilde{u}_j c_{k-j} \quad \text{and} \quad \tilde{c}_k = \sum_{j=1}^{k} -\tilde{u}_j s_{k-j} \quad ,$$

  where $\tilde{v}_j = j \cdot v$; see also [1].

- Additional (optional) nested loops within the inner loop over $k = 1, \ldots, o$ as shown above for $\tilde{s}_k$ and $\tilde{c}_k$, where the loop bounds depend on k.

An obvious target to control loop unrolling in the code generator is the outer loop over the directions. The following reasons make this a good candidate.

- To compute complete tensors up to order o, the number of directions (depending on the number n of inputs to $\boldsymbol{f}$) is $d = \binom{n+o-1}{o}$, that is, d grows quickly with n and o.

- It allows a *uniform split* of the data structure and is relatively easy to implement.

- It is plausible to the user because it is closely related to propagating slices of the seed matrix $\boldsymbol{S}$, a known practice for first-order derivatives with large d.

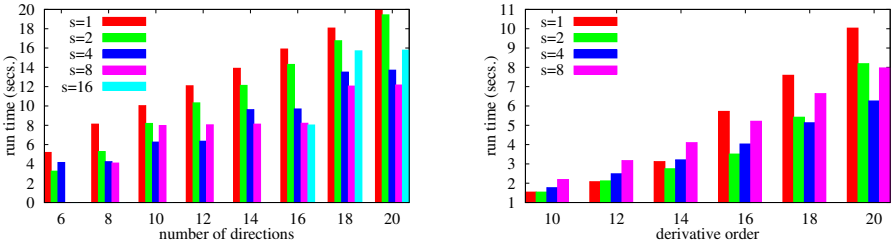
With the above in mind, we modified Rapsodia so that the user can specify to the generator the number s of slices into which d may be split. An example of the resulting code is shown in Fig. 2. In order to ensure a uniform split of

```
1  r.v = a.v * b.v;
2  for(i=0;i<=4;i+=1)
3    {
4      r.s[i].d1_1 = a.v * b.s[i].d1_1 + a.s[i].d1_1 * b.v;
5      r.s[i].d1_2 = a.v * b.s[i].d1_2 + a.s[i].d1_1 * b.s[i].d1_1 + a.s[i].d1_2 * b.v;
6      r.s[i].d1_3 = a.v * b.s[i].d1_3 + a.s[i].d1_1 * b.s[i].d1_2 + a.s[i].d1_2 * b.s[i].
           d1_1 + a.s[i].d1_3 * b.v;
7      r.s[i].d2_1 = a.v * b.s[i].d2_1 + a.s[i].d2_1 * b.v;
8      r.s[i].d2_2 = a.v * b.s[i].d2_2 + a.s[i].d2_1 * b.s[i].d2_1 + a.s[i].d2_2 * b.v;
9      r.s[i].d2_3 = a.v * b.s[i].d2_3 + a.s[i].d2_1 * b.s[i].d2_2 + a.s[i].d2_2 * b.s[i].
           d2_1 + a.s[i].d2_3 * b.v;
10   }
```

Figure 2: Generated code for overloaded `*` operators with o=3, d=10, s=5 for `r=a*b`; see also Fig. 1.

the data structure and generated loops, the generator may internally increase d to a multiple of s. Note that to aid compiler optimization, we retain the flat data structure within each slice `s[i]`, generate the loop with fixed bounds, and in the Fortran version declare the overloaded operators and intrinsics to be `elemental`.

Fig. 3 shows the run time for a test example with a mix of operations but a large portion of nonlinear intrinsics. We vary s and keep either o or d fixed. The run times are given for the Intel C++ compiler using the `-O3` flag. As can be seen, there is a distinct optimal s that depends—as one would expect—on both o and d. Diminished performance for s larger than the optimal value is arguably due to the fact that after a certain point, the body of the loop is too small to retain the optimization gains of loop unrolling. Likewise, for s smaller than the optimal value, run times grow with the loop body because the cache becomes too small.



(a): fixed o=20 (b): fixed d=10

Figure 3: Run times for varying d, s, o compiled with `icpc -O3` .

At least one major argument can be made against our approach, namely, that we slice *without considering the computational complexity of the loop body*. The difference becomes apparent when comparing Fig. 4 with Fig. 2. By offering only the slice number as a control parameter, we split all the direction loops uniformly and consequently obtain loop bodies that still may be too large for the compiler optimization, while other loop bodies become very small. Therefore, for the overall performance one must consider not only o and d but also the percentage of expensive vs. inexpensive operators and intrinsics used by the particular application.

In other contexts such as the empirically optimized BLAS in ATLAS [16] or the loop optimization in Orio [17], the complexity of the loop body plays a central role in determining, for instance, to which extent loops are unrolled. One might expect that gains could be made by using these tools in our context, and we are not trying to suggest otherwise. Instead we are covering a different aspect where the application-specific parameters o, d, and s are fixed a priori by the user and the generator creates the application specific library with fixed loop bounds where possible and a data structure as flat as possible. While one may be able to afford autotuning a generic AD library for a specific application context, it is beyond the scope of this paper to show to what extent the autotuning can

```
1  r.v = a.v + b.v;
2  for(i=0;i<=4;i+=1)
3    {
4      r.s[i].d1_1 = a.s[i].d1_1 + b.s[i].d1_1;
5      r.s[i].d1_2 = a.s[i].d1_2 + b.s[i].d1_2;
6      r.s[i].d1_3 = a.s[i].d1_3 + b.s[i].d1_3;
7      r.s[i].d2_1 = a.s[i].d2_1 + b.s[i].d2_1;
8      r.s[i].d2_2 = a.s[i].d2_2 + b.s[i].d2_2;
9      r.s[i].d2_3 = a.s[i].d2_3 + b.s[i].d2_3;
10   }
```

Figure 4: Generated code for overloaded `+` operators with o=3, d=10, s=5 for `r=a+b`; cf. Fig. 2.

or cannot consistently capture and exploit the information that the Rapsodia generator explicitly uses.

In this sense we see the unrolling discussed here as an improvement to Rapsodia that benefits some applications, without making a statement regarding autotuning tools. The choice of s in practical applications could itself be made empirically but otherwise will be guided by recommendations in the Rapsodia manual.

4 Parallel Derivative Computations

The Taylor coefficient propagation logic in the overloaded operators takes the computed intermediated values of $\boldsymbol{f}$ (i.e., the v_0) as input, but all other data dependencies are among the Taylor coefficients $v_i^k, i = 1, \ldots, o$ themselves, and the propagations for the individual directions are mutually independent; that is, there is no data dependency between v_i^j and v_l^k if $j \neq k$. Consequently, the outer loops over the d directions are easily parallelizable by splitting them into slices. This approach neatly coincides with the limited loop unrolling strategy discussed in Sec. 3. Thus, AD computations should be well suited to exploit the current multicore architectures by distributing the propagation slices across the cores with the goal of reducing the computation time.

4.1 Parallelism with OpenMP

The opportunity to parallelize the outer loop has been well recognized, and various attempts have been made to exploit it, in particular with the help of OpenMP; see [12, 13, 14]. OpenMP is a convenient method to enable shared-memory parallelism. Parallel execution is triggered by directives that are read and interpreted by the compiler, the aim being to relieve the programmer of the arduous task of manually introducing the parallelism, for example, by explicit multithreaded programming. For our particular context one can use OpenMP in two fundamental ways.

Coarse-Grained OpenMP The first is *coarse-grained* parallelism, which uses a single parallel section covering the entire execution of $\boldsymbol{f}$, as was done in, for example, [12]. The outer loop over the d directions is moved from inside the overloaded operators to the driver code (i.e., the code that calls $\boldsymbol{f}$). The code of each overloaded operator contains the loop body for the respective slice into which the directions were split. Consequently the number of slices can then simply be equal to the number of cores used. Of course, one can also retain more than one slice per core and keep a loop over slices in the overloaded operators. For example, if d=16 on a four-core platform, we can compute 4 directions on each core and have per core 2 slices with 2 directions each. This approach requires the OpenMP setup to be done manually in the driver; in other words, the user has to write the logic that first initializes the directions, then calls $\boldsymbol{f}$ in the parallel loop, and finally collects the results. No specific change is needed for the Rapsodia code generator or the preparation of the $\boldsymbol{f}$ source code for overloading. With the Rapsodia library we saw results similar to the ones first presented in [12].

The major drawback of this approach is that it requires the parallel execution of the entire $\boldsymbol{f}$, which at least entails the computation of all the same intermediate values of $\boldsymbol{f}$ on each core. This can become a serious efficiency concern if there are intermediate values in $\boldsymbol{f}$ that do not impact the derivative values of interest[3] but whose computation contributes a sizable portion to the cost of computing the entire $\boldsymbol{f}$. At worst one may be not be able to execute $\boldsymbol{f}$ in multiple concurrent threads. The parallel execution of $\boldsymbol{f}$ requires $\boldsymbol{f}$ to be *side-effect free* or else the side effects will mutually impact the instances of $\boldsymbol{f}$ that are being executed in parallel leading to inconsistent results. For instance, $\boldsymbol{f}$ may update global variables shared among all threads or write output to some file with a hardcoded name, and consequently the parallel runs mutually overwrite the output, leaving an inconsistent result. The latter was the case with some example codes previously used with Rapsodia. Except for trivial $\boldsymbol{f}$ and unless $\boldsymbol{f}$ was written already with parallel execution in mind, it is often difficult to ascertain that $\boldsymbol{f}$ is side-effect free. This poses a significant problem for the practical application of the approach.

Fine-Grained OpenMP Rather than requiring that the user code first be made side-effect free, we tried a second approach that we call *fine-grained* parallelism, also described in [12]. Here, the parallelized loop is still the loop over the directions, but it remains within the overloaded operators. The major advantage of this approach is that almost all changes are within the Rapsodia-generated code, hidden from the user. Done in a naive way, however, this approach incurs significant overhead each time the parallel loop inside the overloaded operators is entered and exited. This overhead is caused by the frequent creation and termination of the OpenMP threads and results in disappointing run times. To avoid this overhead, we followed the suggestion in [12] and used the *orphaning*

[3]These are often called *passive variables* as opposed to active variables whose type is changed to trigger the overloaded operators.

concept that allows the threads to be created once and then kept alive throughout the execution of $\boldsymbol{f}$. This requires the user to wrap $\boldsymbol{f}$ in the driver in OpenMP directives; but aside from that, no further changes need to be made to the driver. However, it again implies parallel execution of $\boldsymbol{f}$ itself and therefore is not usable in cases where $\boldsymbol{f}$ instances cannot run in parallel or where one would like to avoid replicating the computation of the same function values.

Another unfortunate consequence of the orphaning approach is that because $\boldsymbol{f}$ is executed in parallel, the result variable of an overloaded operator is private to each thread. Because each thread computes only a portion of the derivative, after the propagation the threads must exchange their results. We accomplished this exchange within each overloaded operator and intrinsic by storing the computations of each thread in a global placeholder and then copying these data to the private result variable of each individual thread. This copying implies a (different) overhead, and the timing results still are disappointing.

We tested these three parallelization techniques on an 8-core, 64-bit AMD-processor machine. All tests were conducted with the same benchmarking code for C++ and Fortran, using the GNU and Intel compilers. The results point to the fact that OpenMP provides too little control over the threads to be useful for the fine-grained approach. Therefore, we developed another approach using a circular queue, with threads controlled explicitly by the library (rather than the implicit control provided by OpenMP).

4.2 Asynchronous Multithreaded Derivative Computation

As indicated in the beginning of this section, there is a dependency of Taylor coefficients' propagation logic to the $\boldsymbol{f}$ values v_0, but no dependency in the other direction. Consequently, the coefficients may be propagated asynchronously (lagging behind) to $\boldsymbol{f}$. In other words, we can remove the unnecessary synchronizations between the parallel propagation of the slices and the continued computation of $\boldsymbol{f}$ that exist in the fine-grained OpenMP approach. The price for the asynchronicity is a temporary storage of some intermediate values v_0, *operation identifiers*, and *locations* (think pseudo addresses of the program variables) in a circular queue of a predefined size. The queue entries are similar to the concept of the *tape* entries in Adol-C.

Each overloaded operator/intrinsic triggers in the (single threaded) execution of $\boldsymbol{f}$ the writing of a queue entry; see Fig. 5. The queue entries are read by multiple threads, each of which (running on its own core) is responsible for propagating its slice of Taylor coefficients. These threads operate asynchronously from one another, so the writer thread can be ahead of the propagation threads limited only by the queue size. The propagation threads obtain the propagation operation from the queue and operate on the Taylor coefficients stored in thread-specific slices in a *work array*. The aforementioned locations that are part of the queue entries are used as indices into the work array. Like Adol-C, we rely on C++ constructors and destructors to manage the locations for each active program variable. Fortran does not provide a destructor-like concept. Without a hook to trigger the release of a location when a variable goes out of

scope, the size of the work array will often become impractical. Orchestrating this release of locations requires either additional manual source code changes within $\boldsymbol{f}$ or the use of source transformation tools to automatically inject support library calls that trigger location releases for active stack variables at the end of a Fortran function or subroutine and for deallocation calls pertaining to active variables. Neither approach has as of yet been implemented.

Eventually, the driver logic will want to retrieve the Taylor coefficients of the outputs and can do so with calls to `getCoeff()`. The implementation of `getCoeff()` waits until all propagation threads have reached it in the queue and only then retrieves the data from the work array (see Fig. 5). Synchronization happens when the queue is empty (seen separately for each propagation thread) or full or when Taylor coefficients are requested by the driver of $\boldsymbol{f}$.

We implemented two methods for thread synchronization. The first method uses traditional locks from the standard Posix pthreads library. The second method uses atomic operations, such as fetch-and-decrement, from the Open Portable Atomics (OpenPA) library [15]. OpenPA is a library implementing atomic primitives for shared-memory applications. Using atomic operations allows variables to be updated *atomically* without the need for explicit thread serialization to protect the update using pthread locks. In our algorithm, each queue entry contains a shared variable storing information on the work that was done for that entry. With OpenPA, this variable is an integer representing the number of threads that have not computed their slice of the derivative (see the filled circle sections of the queue entries in Fig. 5). This variable is atomically decremented (i.e., marked done) each time a thread completes its computation on that entry. In the pthreads implementation, a bitmap is used, where each thread sets its corresponding bit when it completes its computation. This update is protected by an explicit pthread lock.

We use three spinlocks (shown in red in the flow charts in Fig. 5), rather than condition variables, when waiting for an element to become free (or full) in both the OpenPA and pthread implementations in order to maximize performance. Using spinlocks allows a waiting thread to immediately process the free (or full) entry without the overhead of a system call and context switch associated with condition variables. Generally speaking, because spinlocks utilize the processor core while waiting, this approach may have a negative impact when processor cores are oversubscribed, that is, when more than one thread is scheduled on the same core. In an oversubscribed environment, using condition variables can improve processor utilization by allowing waiting threads to yield the core to other ready threads. In our algorithm, however, we expect that the execution time of the threads will be dominated by the computation of derivative slices or by the evaluation of the function and that little time will be spent waiting on a spinlock. For this reason, there would be no benefit in creating additional threads and oversubscribing the cores, which may in fact reduce the overall performance because of cache invalidations associated with context switching multiple threads on the same core.

One important issue that arose during implementation was that the spinlocks, used to avoid race conditions, were optimized away by the Intel compiler

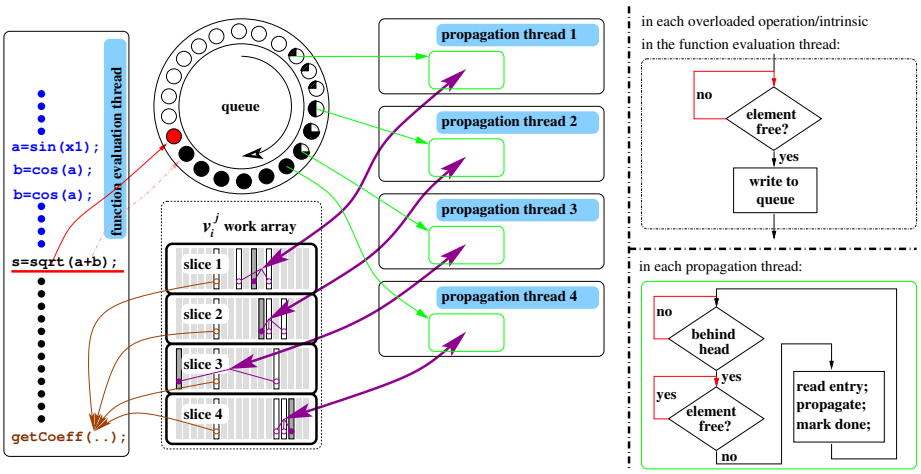


Figure 5: Asynchronous computation with circular queue and flow charts for queue writer and readers.

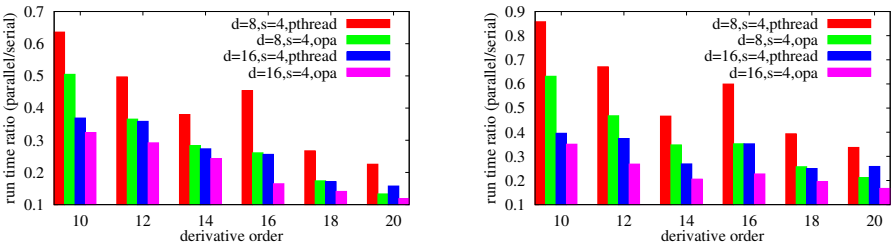
when compiled with `-O3`. We are investigating the cause of this, but in the meantime we have addressed this problem by introducing calls to `nanosleep()` in the body of these loops. This workaround likely had a detrimental impact on the performance results presented here. Well-performing spinlocks are generally useful and may eventually be provided by OpenPA. Fig. 6 shows the runtime ratios of the serial execution over the two queue implementations. On the abscissas we show the varying o values. All four plots show that, for sufficiently large o and d, the queue implementation, despite the overhead of writing/reading the queue and managing the pseudo addresses for the work array, becomes a practical alternative to the serial approach. As one would expect, when optimization is turned on, the o and d values for which the queue approach becomes viable are larger. Even for this implementation prototype, however, they remain within the range of the applications for which Rapsodia is intended.

We view the current implementation as a usable proof-of-concept and expect further improvements to make the queuing approach useful for smaller values of d and o than those shown in, for example, Fig. 6(d).

5 Summary

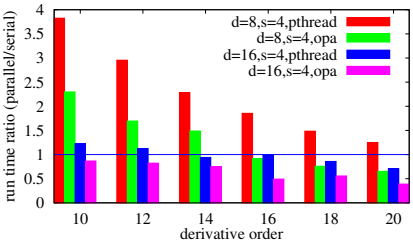
We presented two avenues for modifying the Rapsodia-generated overloading libraries. In the purely serial case the introduction of a limit to the unrolling of the outer loop was shown to be beneficial for sufficiently large derivative order and number of directions. The observed runtime improvements for the serial execution reach as high as 50%; see Fig. 3.

To exploit multicore hardware, we investigated different approaches to parallelize the derivative computation with Rapsodia. Using OpenMP, we observed

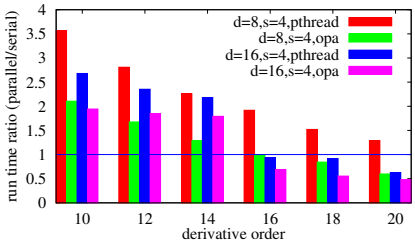


(a): compiled with `g++ -O0`

(b): compiled with `icpc -O0`



(c): compiled with `g++ -O3`



(d): compiled with `icpc -O3`

Figure 6: Runtime ratios plotted over varying o for combinations of (d, s, implementation) and compiler/optimization levels set up for 4 propagation threads and executed on an 8-core, 64-bit AMD-processor machine.

good results similar to those of previous studies for an approach that requires the user to explicitly parallelize the execution of the target function $\boldsymbol{f}$. This approach does not require changes to the Rapsodia code generator; instead, all modification work has to be done by the user. Because $\boldsymbol{f}$ is executed in multiple concurrent threads it must be side-effect free, which may not always be the case. Even if it is side-effect free, the efficiency can be can severely diminished because the same function values are computed multiple times. Alternative approaches with OpenMP that attempt to address these concerns imply a substantial overhead resulting in disappointing run times and therefore have been abandoned.

We introduced an alternative approach that uses explicit multithreaded programming to enable the asynchronous parallel computation of the Taylor coefficients. Here, the model $\boldsymbol{f}$ itself is not executed in parallel and we therefore do not require it to be side-effect free. While the implementation is still in the proof-of-concept stage, these timing results demonstrate the strength of the asynchronous approach for computing $\boldsymbol{f}$ with higher d and o.

The results also demonstrate the superiority of the OpenPA-aided implementation compared to the implementation that has to rely on only the pthread library. The OpenPA implementation avoids some overhead incurred with pthread interfaces by using atomic hardware operations to modify any shared data.

While we were analyzing the performance of the current implementation,

the cost of frequently locking and unlocking pthread locks became especially apparent for the logic that guards the propagation threads operations against the dynamic reallocation of the work array. The current prototype lacks the logic required to safely reallocate the work array to a different address without locks, but this will be added in the near future.

The notable absence of Fortran results for the asynchronous approach has already been explained by the lack of a destructor in Fortran. The suggested alternatives will be pursued in future work.

Additional future work entails further optimizations to the queue implementation, as well as testing on more platforms, e.g. the Blue Gene/P and the next generation Intel and AMD CPUs with an increased number of cores.

Acknowledgments We thank Paul Hovland for valuable hints to Alexis Malozemoff during his internship at Argonne in the summer of 2009. This work was supported by the U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

[1] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd Edition, no. 105 in Other Titles in Applied Mathematics, SIAM, Philadelphia, PA, 2008.
URL http://www.ec-securehost.com/SIAM/OT105.html

[2] H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, B. Norris (Eds.), Automatic Differentiation: Applications, Theory, and Implementations, Vol. 50 of Lecture Notes in Computational Science and Engineering, Springer, New York, NY, 2005. doi:10.1007/3-540-28438-9.

[3] C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Vol. 64 of Lecture Notes in Computational Science and Engineering, Springer, Berlin, 2008. doi:10.1007/978-3-540-68942-3.

[4] AD community website, http://www.autodiff.org.

[5] A. Griewank, D. Juedes, J. Utke, Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++, ACM Transactions on Mathematical Software 22 (2) (1996) 131–167.
URL http://doi.acm.org/10.1145/229473.229474

[6] J. D. Pryce, J. K. Reid, ADO1, a Fortran 90 code for automatic differentiation, Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England (1998).
URL ftp://ftp.numerical.rl.ac.uk/pub/reports/prRAL98057.pdf

[7] A. Griewank, J. Utke, A. Walther, Evaluating higher derivative tensors by forward propagation of univariate Taylor series, Mathematics of Computation 69 (2000) 1117–1130.

[8] Rapsodia, http://www.mcs.anl.gov/Rapsodia/.

[9] I. Charpentier, J. Utke, Fast higher-order derivative tensors with Rapsodia, Optimization Methods Software 24 (1) (2009) 1–14. doi:10.1080/10556780802413769.

[10] I. Charpentier, J. Utke, Rapsodia: User manual, Tech. rep., Argonne National Laboratory, latest version available online at http://www.mcs.anl.gov/Rapsodia/userManual.pdf.

[11] I. Charpentier, C. D. Cappello, J. Utke, Efficient higher-order derivatives of the hypergeometric function, in: Bischof et al. [3], pp. 127–137. doi:10.1007/978-3-540-68942-3_12.

[12] H. M. Bücker, B. Lang, D. an Mey, C. H. Bischof, Bringing together automatic differentiation and OpenMP, in: Proceedings of the 15th ACM International Conference on Supercomputing, Sorrento, Italy, June 17–21, 2001, ACM Press, New York, 2001, pp. 246–251. doi:10.1145/377792.377842.
URL http://doi.acm.org/10.1145/377792.377842

[13] H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, D. an Mey, Explicit loop scheduling in openmp for parallel automatic differentiation, in: J. N. Almhana, V. C. Bhavsar (Eds.), Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002, IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 121–126. doi:10.1109/HPCSA.2002.1019144.
URL http://doi.ieeecomputersociety.org/10.1109/HPCSA.2002.1019144

[14] H. M. Bücker, A. Rasch, A. Wolf, A class of openmp applications involving nested parallelism, in: Proceedings of the 19th ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14–17, 2004, Vol. 1, ACM Press, New York, 2004, pp. 220–224. doi:10.1145/967900.967948.
URL http://doi.acm.org/10.1145/967900.967948

[15] Open source Portable Atomics library (openPA), http://trac.mcs.anl.gov/projects/openpa.

[16] Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/.

[17] Orio: An Annotation-Based Empirical Performance Tuning Framework, http://trac.mcs.anl.gov/projects/performance/wiki/Orio.